Project 2 Report: Simlarity Join

在Postgresql源码中实现Jaccard index和Levenshtein distance

系统与源码理解

需修改的有如下三个文件

- `/src/include/catalog/pg_proc.dat`: add the prototype of user-defined functions, which will generate `fmgroids.h`, `fmgrprotos.h`, and `fmgrtab.c` by `Gen_fmgrtab.pl` when compiling.
- `/src/backend/utils/adt/similarity_join.c`: our user-defined functions
- `/src/backend/utils/adt/Makefile`: add support to compile similarity_join.c

我写了一个`patch.py`以便一键完成以上修改或去修改，详见`README.md`

思路和算法分析

Levenshtein Distance

令字符串A的前i个字符为字符串A'，字符串B的前j个字符为字符串B'

则从A'「编辑」成B'的距离为

$$d(i,j) = \min(d(i-1,j)+1, d(i,j-1)+1, d(i-1,j-1)+[a_i \neq b_j])$$

其中$[a_i \neq b_j] = \begin{cases} 0, & \text{i-th character of string A is equal to j-th character of string B} \\ 1 & \text{otherwise} \end{cases}$

用动态规划的方法求解，复杂度$O(m \cdot n)$，m和n分别是A和B所含的字符个数

当已知要求「编辑距离」小于某一值时，算法可以提前终止，因$\begin{cases} d(i,j) \geq d(i-1,j) \\ d(i,j) \geq d(i,j-1) \\ d(i,j) \geq d(i-1,j-1) \end{cases}$，且在DP过程中i, j是递增的

Jaccard Index

令字符串A和字符串B的Jaccard指数定义如下

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|} = \frac{|A \cap B|}{|A| + |B| - |A \cap B|}$$

交集的计算可用hash实现，计算Jaccard指数的复杂度为$O(m+n)$，m和n分别是A和B所含的字符个数

关键代码说明

- `func/similarity_join.c`: 两个函数的实现的代码在
- `patch.py`: 一键修改源码
- `sql/`: 建表、create/drop function、测试等SQL语句

similarity_join.c

- hash table: `hash_init()`, `hash_insert(char a, char b)`, `hash_delete(char a, char b)`
- Optimized Levenshtein distance:

```
/* Optimized Levenshtein distance by early stop */
/*
    if Levenshtein distance between a and b is less than k
        return 1
    else
        return 0
*/
static int _levenshtein_distance_less_than(const char *a, const char *b, const int len_a, const int len_b, int k)
{
    static int d[MAX_LEN][MAX_LEN];
    d[0][0] = 0;

    /* Initialize the first row and column */
    for (int i = 1; i <= len_b; i ++)
        d[i][0] = i;
    for (int j = 1; j <= len_a; j ++)
        d[0][j] = j;

    for (int i = 1; i <= len_b; i ++)
    {
        int all_ge_k = 1;//All greater or equal to k
        for (int j = 1; j <= len_a; j ++)
        {
            d[i][j] = MIN3(d[i-1][j] + 1, d[i][j-1] + 1, d[i-1][j-1] + (a[j-1] == b[i-1] ? 0 : 1));
            if (d[i][j] < k) all_ge_k = 0;
        }
        if (all_ge_k)
            return 0;
    }
    return d[len_b][len_a] < k;
}
```

- Jaccard Index: return 2-gram jaccard index of

```
/* Hash table: much faster than naive look-up table */
/* return a float, which is the jaccard index of string a and b */
static float _jaccard_index(const char *a, const char *b, const int len_a, const int len_b)
```

```
{
    int intersect_cnt = 0;

    hash_init();

    /* Counting 2-gram of a by inserting into hash table */
    hash_insert('$', a[0]);
    for (int i = 1; i < len_a; i ++)
        hash_insert(a[i-1], a[i]);
    hash_insert(a[len_a-1], '$');

    /* Counting intersection search for and delete 2-gram of b */
    if (hash_delete('$', b[0])) intersect_cnt ++;
    for (int i = 1; i < len_b; i ++)
        if (hash_delete(b[i-1], b[i]))
            intersect_cnt ++;
    if (hash_delete(b[len_b-1], '$')) intersect_cnt ++;

    return (float)intersect_cnt / (len_a + len_b + 2 - intersect_cnt);
}
```

patch.py

修改下三个文件，若运行时加上`-u`或`--unpatch`，则为恢复原先代码

- `modify_dat`: 在`/src/include/catalog/pg_proc.dat`的最后加上自定义的函数及其信息，如下

```
# similarity join functions begin
{ oid => '9997', descr => 'jaccard index',
  proname => 'jaccard_index', prorettype => 'float8', proargtypes => 'text text',
  prosrc => 'jaccard_index' },
{ oid => '9998', descr => 'levenshtein distance',
  proname => 'levenshtein_distance', prorettype => 'int4', proargtypes => 'text text',
  prosrc => 'levenshtein_distance' },
{ oid => '9999', descr => 'whether levenshtein distance is less than threshold',
  proname => 'levenshtein_distance_less_than', prorettype => 'bool', proargtypes =>
'text text int4',
  prosrc => 'levenshtein_distance_less_than'
},
# similarity join functions end
```

- 将实现的`similarity_join.c`复制到`/src/backend/utils/adt/`下
- `modify_makefile`: 在`/src/backend/utils/adt/Makefile`中添加对于`similarity_join.c`的编译

实验结果

复现结果的方法见`README.md`

- lev: Levenshtein distance
- opt-lev: optimized (less-than-k) Levenshtein distance

时间

	1(opt-lev/lev)	2(opt-lev/lev)	3(opt-lev\lev)	4(jaccard)	5(jaccard)	6(jaccard)
Time(s)	2.183/3.621	2.354/10.367	4.373/19.384	2.040	3.236	3.824

结果（计算时字母全部小写化）

	1(opt-lev)	2(opt-lev)	3(opt-lev)	4(jaccard)	5(jaccard)	6(jaccard)
Count	3252	2130	2592	1488	2320	2308

优化

gcc -O2

效率提高一倍以上

Levenshtein Distance: Early Stop

在已知条件是「编辑距离」小于某个阈值的时候，DP计算时可以提前知道条件不符合，函数可以提前终止

在本次实验中，效率显著提高2-4倍

Jaccard Index: Hashing

在计算交集的时候使用Hash，如下

- 对A的每个2-gram，插入哈希表
- 对于B每个2-gram，若哈希表中存在此元素，删之，并且交集个数加一